A Programmable SoC-Based Accelerator for Privacy-Enhancing Technologies and Functional Encryption

Bahadori, Milad

2020-10

Bahadori , M & Jarvinen , K 2020 , ' A Programmable SoC-Based Accelerator for Privacy-Enhancing Technologies and Functional Encryption ' , IEEE Transactions on Very Large Scale Integration Systems , vol. 28 , no. 10 , pp. 2182-2195 . https://doi.org/10.1109/TVLSI.2020.3010585

http://hdl.handle.net/10138/325173
https://doi.org/10.1109/TVLSI.2020.3010585

acceptedVersion

A Programmable SoC Based Accelerator for Privacy Enhancing Technologies and Functional Encryption

Milad Bahadori and Kimmo Järvinen

***Abstract*—A multitude of privacy enhancing technologies have been presented recently to solve privacy problems of contemporary services utilizing cloud computing. Many of them are based on additively homomorphic encryption that allows computation of additions on encrypted data. The main technical obstacles for adaptation of privacy enhancing technologies in practical systems are related to performance overheads compared to current privacy-violating alternatives. In this paper, we present a HW/SW codesign for programmable SoCs that is designed for accelerating applications based on Paillier encryption. Our implementation is a microcode based multi-core architecture which is suitable for accelerating various privacy enhancing technologies using additively homomorphic encryption with large integer modular arithmetic. We instantiate the implementation in a Xilinx Zynq-7000 programmable SoC and provide performance evaluations in real hardware. We also investigate its efficiency in a high-end Xilinx UltraScale+ programmable SoC. We evaluate the implementation with two target use cases that have relevance in privacy enhancing technologies: privacy-preserving computation of squared Euclidean distances over encrypted data and multi-input functional encryption for inner products. Both of them represent the first hardware acceleration results for such operations and, in particular, the latter one is among the very first published implementation results of functional encryption on any platform.**

***Index Terms*—SoC, FPGA, homomorphic encryption, Paillier encryption, privacy-preserving, functional encryption.**

I. Introduction

THE ever-growing use of cloud computing has moved much of users' sensitive data into service providers' (e.g., Amazon, Facebook, Google, Microsoft, etc.) servers where the data is both stored and processed, and this trend is only expected to accelerate in the future thanks to the emerge of the Internet-of-Things (IoT). Service providers are known to regularly exploit users' data, e.g., for marketing purposes, and it is evident that users' privacy has decreased. Many privacy problems could be avoided by encrypting the data before storing it into the cloud but this prevents cloud computing because processing of encrypted data becomes impossible. The cryptography community has worked on this problem already for a long time (see, e.g., [1]) and many types of solutions addressing different aspects of these problems have been proposed: e.g., homomorphic encryption [2], attribute-based encryption [3], searchable encryption [4], and functional encryption [5], just to name a few. In this paper, we will show how the performance overhead of such solutions can be reduced by designing an efficient accelerator on a programmable System-on-Chip (SoC) that combines processor cores with Field Programmable Gate Array (FPGA) resources.

Partially homomorphic encryption allows computing a limited set of operations over encrypted data. We focus on Additively Homomorphic Encryption (AHE) and Paillier encryption [6] in particular that allows additions with encrypted data. Even such a limited functionality enables applications that can go a long way as shown by multiple publications and discussed in more detail below. The benefit of AHE compared to Fully Homomorphic Encryption (FHE) [2], which allows arbitrary computations on encrypted data, or Somewhat Homomorphic Encryption (SHE), which allows arbitrary computations up to some complexity limit, is that AHE schemes are a lot simpler and typically lead to more efficient outcomes. Despite this, even AHE introduces significant performance penalties compared to performing the same operations on unencrypted data. Thus, there is a clear need for hardware acceleration of these operations, but very little work exists on this topic in the literature. An FPGA-based accelerator for Paillier encryption was presented by San et al. in [7] in 2016, but no other works seems to be available. This is somewhat surprising given that a large body of work is available on accelerating FHE/SHE (see, e.g., [8], [9], [10], [11], [12], [13], [14], [15]), which are arguably much further from practical adaptation than AHE.

Over the years, Paillier encryption has been a central building block in many Privacy Enhancing Technologies (PETs). Paillier encryption was also recently standardized by ISO [16] making it a very attractive alternative for practitioners because few other AHE schemes have been standardized. Examples of use cases of Paillier encryption for PETs include biometrics with fingerprints and face recognition (see, e.g., [17], [18], [19], [20]), smart grid related energy-metering and statistics (see, e.g., [21], [22], [23]), indoor localization (see, e.g., [24], [25], [26]), medical data processing (see, e.g., [27]), and different types of user matching applications for social media, job seeking or dating services (see, e.g., [28], [29]). In these schemes, a user typically encrypts inputs using Paillier encryption and a server uses the homomorphic property on the ciphertexts together with its own input to the scheme (e.g., a database) to compute results without revealing the users' inputs. A particularly popular operation in these schemes has been the computation of Squared Euclidean Distances (SEDs) between the user's input and the server's database (see, e.g., [17], [19], [20], [24], [25], [28]). We focus on SEDs in this paper in order to demonstrate the usability of our implementation for accelerating PETs.

Another use case of AHE that also relates to PETs can be found in Functional Encryption (FE) [5], [30], which is a novel paradigm of encryption that allows to evaluate a function over

M. Bahadori and K. Järvinen are with the Department of Computer Science, University of Helsinki, FI-00014 Helsinki, Finland.
E-mail: {milad.bahadori, kimmo.u.jarvinen}@helsinki.fi
Date of the manuscript: January 25, 2021.

encrypted data while obtaining no other information except the result of the function. Recently, Paillier encryption has been used for realizing different types of efficient FE schemes in [31], [32]. In this paper, we show that such schemes can be accelerated with our implementation. Also other advanced schemes such as searchable encryption (see, e.g., [33]) utilize AHE and could benefit from our design, but they are not directly considered in this work. In this paper, we present the following contributions:

- We present a programmable SoC based architecture for accelerating large integer modular arithmetic that has been tailored for fast computation of AHE (and, particularly, Paillier encryption). The accelerator utilizes a microcode based architecture which provides flexibility and allows using it for different types of applications. It uses a multi-core structure that allows exploiting the inherent parallelism that is available in many use cases of AHE. To the best of our knowledge, this is the first published accelerator for applications enabled by Paillier encryption; the main focus of [7] was on encryption and decryption operations of Paillier and the acceleration of a PET was studied only cursorily.
- We instantiate the proposed architecture as a Hardware/Software (HW/SW) codesign implemented in a Xilinx Zynq-7000 programmable SoC and provide performance evaluations with real hardware for two types of use cases: Firstly, we use the implementation for computing SEDs over encrypted data, which has been used in multiple PETs as discussed above. This is the first published work that is capable of accelerating various types of PETs and shows that significant speed improvements can be achieved, consequently, increasing the practical adaptability of these PETs. Secondly, we use the design for accelerating an FE scheme for inner products from [31], [32] that is based on Paillier encryption. This represents the first hardware-based accelerator for FE and shows that SoC-based acceleration can bring this promising new concept to practical feasibility.

The rest of this paper is organized as follows: Sect. II surveys the preliminaries of Paillier encryption and the two use cases that we consider in this paper. Sect. III describes the specific algorithms that are implemented. Sect. IV presents the architecture of the HW/SW codesign and discusses its instantiation in the target SoCs. Sect. V presents the results and analysis and, finally, Sect. VI ends the paper by drawing conclusions and identifying certain directions for future work.

II. Preliminaries

A. Additively Homomorphic Encryption

Here, we will discuss AHE and formally introduce Paillier encryption [6]. Let $(\!|m|\!)$ denote the ciphertext containing m and let $\mathsf{Enc}(pk, m)$ and $\mathsf{Dec}(sk, (\!|m|\!))$ denote the encryption and decryption with public key pk and secret key sk, respectively. An AHE scheme allows computing additions with ciphertexts. Namely, when given two ciphertexts $(\!|m_1|\!) = \mathsf{Enc}(pk, m_1)$ and $(\!|m_2|\!) = \mathsf{Enc}(pk, m_2)$, the following property holds:

$$\mathsf{Dec}(sk, (\!|m_1|\!) \circ (\!|m_2|\!)) = m_1 + m_2. \tag{1}$$

I.e., $(\!|m_1 + m_2|\!)$ can be computed by applying the operation $\circ$ on $(\!|m_1|\!)$ and $(\!|m_2|\!)$. It is noteworthy that despite being limited to additions, an AHE still allows to compute multiplications by a scalar t by repetitively applying the operation $\circ$:

$$\mathsf{Dec}(sk, \underbrace{(\!|m|\!) \circ (\!|m|\!) \circ \ldots \circ (\!|m|\!)}_{t \text{ times}}) = t \cdot m. \tag{2}$$

1) Paillier Encryption: The Paillier public-key encryption scheme [6] is a probabilistic encryption scheme based on the decisional composite residuosity problem. Paillier encryption comprises the following three algorithms:

- **Key Generation.** Given a security parameter κ (e.g., $\kappa = 2048$), choose two random primes p and q of the length $\kappa/2$ and compute $N = p \cdot q$. Also, select a group generator g for the multiplicative group $\mathbb{Z}^*_{N^2}$ such that the order of g is a non-zero multiple of N. The public key pk is the tuple (N, g) and the secret key sk is $\lambda = \text{lcm}(p-1, q-1)$.
- **Encryption.** Take a message $m \in \mathbb{Z}_N$ and a public key (N, g) as inputs and select a random $r \in_R \mathbb{Z}^*_N$. Then, compute and return the ciphertext $(\!|m|\!)$:

$$\mathsf{Enc}(pk, m) = g^m \cdot r^N \bmod N^2. \tag{3}$$

- **Decryption.** Take $(\!|m|\!) \in \mathbb{Z}^*_{N^2}$ and the secret key λ as inputs. Then, compute and return the plaintext m:

$$\mathsf{Dec}(sk, (\!|m|\!)) = \frac{L((\!|m|\!)^\lambda \bmod N^2)}{L(g^\lambda \bmod N^2)} \bmod N, \tag{4}$$

  such that $L(u) = \frac{u-1}{N}$ and $L(g^\lambda \bmod N^2)^{-1} \bmod N$ can be precomputed.

For Paillier, the operation $\circ$ is a multiplication. Thus, a homomorphic addition is simply a multiplication of two ciphertexts modulo N^2 and a homomorphic scalar multiplication is an exponentiation to the exponent t modulo N^2 (see (1) and (2)).

B. Use Cases of Paillier Encryption

We consider two specific use cases of Paillier encryption that are relevant in the cloud computing scenario that was discussed in Sect. I. Firstly, we discuss privacy-preserving computation of SEDs between the user's query and the service provider's database, which has multiple use cases in PETs. Secondly, we discuss FE that is a new paradigm for encryption that permits the service provider to compute only a specific function (in our case an inner product) over users' inputs.

1) Privacy-Preserving SEDs over Encrypted Data: Let $\mathbf{Y}$ be a database of n vectors of length m: $\mathbf{Y} = (\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_{n-1})$, where $\mathbf{y}_i = (y_{i,0}, y_{i,1}, \ldots, y_{i,m-1})$ with $y_{i,j} \in \mathbb{Z}_\ell$. Without loss of generality, we assume that ℓ is a power of two and use $\ell = 2^8$ as an example in this paper. Let $\mathbf{x} = (x_0, x_1, \ldots, x_{m-1})$ with $x_i \in \mathbb{Z}_\ell$ be a query vector. We assume that $\mathbf{Y}$ is in the possession of a server and $\mathbf{x}$ is a query sent by a user, who encrypts x_i using pk to keep $\mathbf{x}$ confidential: $(\!|\mathbf{x}|\!) = ((\!|x_0|\!), (\!|x_1|\!), \ldots, (\!|x_{m-1}|\!)) = (\mathsf{Enc}(pk, x_0), \mathsf{Enc}(pk, x_1), \ldots, \mathsf{Enc}(pk, x_{m-1}))$.

The server's task is to calculate encrypted distances $(\!|d_i|\!)$ between the encrypted query $(\!|\mathbf{x}|\!)$ and each vector $\mathbf{y}_i$ in $\mathbf{Y}$. The final target is to find the indices of the k smallest distances, i.e., to find the k nearest neighbors (kNN), but only the

distance calculation phase is done in the encrypted domain. An additional requirement is that also the server wants to keep $\mathbf{Y}$ secret from the user requiring a second stage after the distance calculation phase that finds kNN but prevents the user from receiving the actual distances, which would reveal information about $\mathbf{Y}$ to the user. This can be achieved, e.g., by masking the distances with random masks so that removing the masks and finding the kNN is done with the help of Yao's garbled circuits [34] as shown in [20]. Here, we omit the details about this phase and focus on the computationally more demanding distance calculation phase. It also helps if the server packs several distances into a single ciphertext before sending them to the user because then the communication overhead is reduced. The above scenario is depicted in Fig. 1(a).

a) Squared Euclidean Distances: The SED is a particularly meaningful distance metric in the above setting because it has been used, e.g., for privacy-preserving fingerprint matching [17], face recognition [19], [20], indoor localization [24], [25], and user matching [28]. It can be decomposed into three terms in the following way [19]:

$$d_i = ||\mathbf{x} - \mathbf{y}_i||^2 = \sum_{j=0}^{m-1}(x_j - y_{i,j})^2 = \sum_{j=0}^{m-1} x_j^2 + \sum_{j=0}^{m-1}(-2x_j y_{i,j}) + \sum_{j=0}^{m-1} y_{i,j}^2 = \Delta_{i,1} + \Delta_{i,2} + \Delta_{i,3}. \tag{5}$$

We see that the term $\Delta_{i,1}$ depends only on the user's inputs and $\Delta_{i,3}$ only on the server's inputs and, therefore, they can be computed over the plaintexts and, then, encrypted with pk by the user and server, respectively. Because $\Delta_{i,1}$ is the same for all distances d_i, it suffices for the user to send only one value $(\!|\Delta_1|\!)$. The middle term is calculated so that the user sends $(\!|-2x_j|\!)$ for $j = 0, \ldots, m-1$. The server then utilizes the properties of (1) and (2) to obtain $(\!|\Delta_{i,2}|\!)$:

$$(\!|\Delta_{i,2}|\!) = \prod_{j=0}^{m-1} (\!|-2x_j|\!)^{y_{i,j}} \pmod{N^2}. \tag{6}$$

Finally, the server uses (1) to add the $(\!|\Delta_1|\!)$, $(\!|\Delta_{i,2}|\!)$, and $(\!|\Delta_{i,3}|\!)$:

$$(\!|d_i|\!) = (\!|\Delta_1|\!) \cdot (\!|\Delta_{i,2}|\!) \cdot (\!|\Delta_{i,3}|\!) \pmod{N^2}. \tag{7}$$

b) Ciphertext Packing: Given upper bounds for n, x_j, and $y_{i,j}$, it is possible to calculate an upper bound for the values of d_i. For simplicity, we assume that $d_i \in \mathbb{Z}_{\ell'}$ so that ℓ' is a power of two. If ℓ' is significantly smaller than the plaintext space of the encryption system, then several d_i fit into one ciphertext by placing each into a slot of at least $\lceil \log_2(\ell') \rceil$ bits [20]. The condition holds in particular for Paillier system where the plaintext space is $\mathbb{Z}_N$ with a large N. Thus, this packing can significantly reduce the communication bandwidth when sending $(\!|d_i|\!)$ (or other results of computations with encrypted data) to the user. The server packs T encrypted distances $(\!|d_i|\!), (\!|d_{i+1}|\!), \ldots, (\!|d_{i+T-1}|\!)$ by computing:

$$(\!|C_{\text{pack}}|\!) = \prod_{j=0}^{T-1} (\!|d_{i+j}|\!)^{\ell' j} \pmod{N^2}. \tag{8}$$

As a result, the server sends only one ciphertext ($\lceil \log_2(N^2) \rceil$ bits) instead of T ciphertexts ($T \cdot \lceil \log_2(N^2) \rceil$ bits). After decrypting $(\!|C_{\text{pack}}|\!)$ by sk, the user obtains the plaintext $C_{\text{pack}} = \sum_{j=0}^{T-1}(d_{i+j} \cdot \ell'^j)$ and can unpack $d_i, d_{i+1}, \ldots, d_{i+T-1}$.

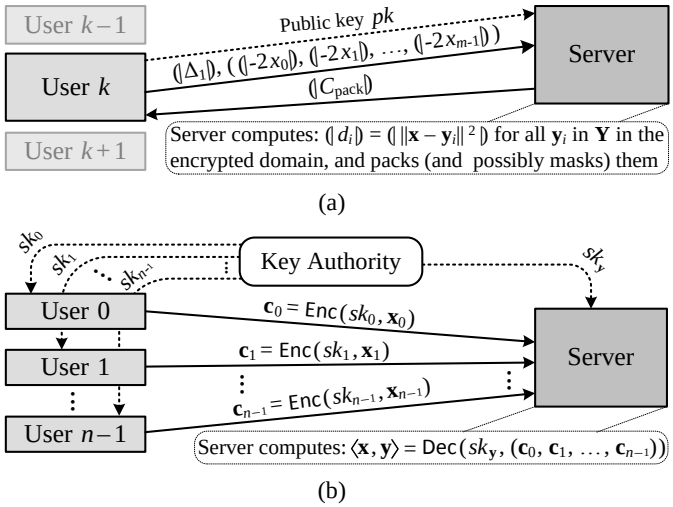

Fig. 1. (a) Privacy-preserving SEDs over encrypted data, (b) MIFE-IP scheme.

2) Multi-Input Functional Encryption for Inner Products: Traditional encryption is "all-or-nothing" in the sense that the holder of the secret key sk obtains the entire plaintext x from the ciphertext $(\!|x|\!)$ and the others get nothing at all. FE [5], [30] is a novel paradigm of encryption that goes beyond this limitation by providing more fine-grained control. Namely, FE allows a key authority (the owner of a master secret key msk) to derive a decryption key sk_f that allows to learn the value of $f(x)$ from $(\!|x|\!)$, but nothing else about x. E.g., FE allows calculating certain statistics over the $(\!|x|\!)$ without revealing x.

MIFE [35], [36] contains n slots and allows n different users to encrypt their own plaintexts $\mathbf{x}_i = (x_{i,0}, x_{i,1}, \ldots, x_{i,m-1})$. The decryption key sk_f permits to compute the value $f(\mathbf{x})$ for $\mathbf{x} = (\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{n-1})$. This makes the scheme very useful for many practical applications such as privacy-preserving data mining and delegated data processing because data can be collected from multiple users. This setup is shown in Fig. 1(b).

While FE constructions for arbitrary polynomial-sized circuits exist (e.g., [37], [38]), they are far from being practical. Here, we focus on FEs that have been designed with efficiency in mind for a limited, but still practically relevant functionality of computing inner product $\langle \mathbf{x}, \mathbf{y} \rangle$ where $\mathbf{x}$ is the user's input vector and $\mathbf{y}$ is the function that is permitted to be computed from $(\!|\mathbf{x}|\!)$. MIFE-IP allows the holder of sk_f to compute

$$f_{\mathbf{y}}(\mathbf{x}) = \sum_{i=0}^{n-1} \langle \mathbf{x}_i, \mathbf{y}_i \rangle = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} x_{i,j} y_{i,j}. \tag{9}$$

In the recent years, many schemes for (multi-input) FE for inner products have been presented based on various cryptographic assumptions and features [31], [32], [39], [40], [41], [42]. We focus on the recent work from Abdalla et al. in [32] that gives a generic construction for building a MIFE-IP scheme from any single-input FE scheme without pairings. We consider the instantiation from [32] that uses the single-input FE for inner products based on Paillier encryption introduced by Agrawal et al. in [31]. The scheme by Agrawal

Algorithm 1: Computation of encrypted middle-terms $(\!|\Delta_{i,2}|\!)$ for SEDs: **(a)** straightforward and **(b)** optimized

Input: Database $\mathbf{Y}$, where $y_{i,j} \in \mathbb{Z}_\ell$, encrypted query $((\!|-2x_0|\!), \ldots, (\!|-2x_{m-1}|\!))$
Output: Middle-terms $((\!|\Delta_{0,2}|\!), \ldots, (\!|\Delta_{n-1,2}|\!))$
$(t_0, t_1, \ldots, t_{n-1}) \leftarrow (1, 1, \ldots, 1)$
for $i = 0$ **to** $n-1$ **do** // **(a) Straightforward**
  for $j = 0$ **to** $m-1$ **do**
    $u \leftarrow \mathrm{ME}((\!|-2x_j|\!), y_{i,j}, N^2)$
    $t_i \leftarrow \mathrm{MM}(t_i, u, N^2)$
for $j = 0$ **to** $m-1$ **do** // **(b) Optimized**
  $u \leftarrow 1$
  for $k = 1$ **to** $\ell - 1$ **do**
    $u \leftarrow \mathrm{MM}(u, (\!|-2x_j|\!), N^2)$
    for $\{i \mid y_{i,j} = k\}$ **do** /* $i \in [0, n-1]$ */
      $t_i \leftarrow \mathrm{MM}(t_i, u, N^2)$
return $((\!|\Delta_{0,2}|\!) = t_0, \ldots, (\!|\Delta_{n-1,2}|\!) = t_{n-1})$

et al. [31] does not use Paillier encryption directly as defined in [6] or in Sect. II-A1, but rather it uses a variation of these algorithms. However, it also relies on the AHE property and is based on the decisional composite residuosity problem. For the sake of brevity, we skip many of the details here and refer interested readers to [31] for details, but we provide the relevant algorithms later in Sect. III.

III. Algorithms

A. Long Integer Modular Arithmetic

Typically, the main performance bottleneck in hardware implementations of public-key cryptography arise from long integer modular arithmetic operations and, especially, from Modular Multiplication (MM). We base our modular operations on the Montgomery modular arithmetic [43] that is widely adopted in the literature for implementations of public-key cryptography. We use algorithms for Modular Addition (MA) and Modular Subtraction (MS), Montgomery Reduction (MR), radix-2^k Montgomery MM, and left-to-right Modular Exponentiation (ME) from [43], [44], [45]. The implementations of these algorithms in our architecture are Constant Time (CT) for all other algorithms except for the left-to-right ME. If timing attacks are considered as a threat, then a CT ME algorithm such as the left-to-right square-and-multiply-always ME can be used via a simple microcode upgrade. The security model is discussed more closely in Sect. IV-D.

B. Squared Euclidean Distances

In the case of SEDs, we focus on the server's computational load because users simply encrypt their inputs using normal Paillier encryptions described in Sect. II-A1; these encryptions are also trivial to parallelize as each vector element can be encrypted independently. For the server, the most significant computational load is related to computing the middle-terms $(\!|\Delta_{i,2}|\!)$ for $i = 0, \ldots, n-1$ using (6) and to computing the ciphertext packings using (8).

Alg. 1(a) describes a straightforward scheme for computing the middle-terms for all vectors $\mathbf{y}_i$ in the server's database $\mathbf{Y}$ individually. This algorithm has the advantage that it is very simple from the control point-of-view and it is straightforward to compute in parallel: the iterations of the outer for loop are distributed to different cores. But, computational resources are wasted because the MEs for computing u are performed multiple times if $y_{i,j} = y_{i',j}$ for $i \neq i'$. An optimization would be to precompute these MEs, but this quickly becomes unfeasible when m grows because of large memory requirements.

Alg. 1(b) shows an optimized algorithm for computing the middle-terms. It processes each element of the user's input separately and scans the corresponding values in all vectors in the server's database $\mathbf{Y}$ in one iteration. This way the MEs of the values in the user's input are computed only once. The disadvantage now is the more complicated control, parallelization, and memory management to handle the variables t_i for the middle-terms if n is large. The algorithm relies on the observation that the values in the server's database (and also in the user's input) are typically relatively small (i.e., ℓ is small). Thus, we make a scan with a value k from 1 to $\ell - 1$ (or to $\max(y_{i,j})$) for a particular j and multiply the corresponding $u = (\!|x_j|\!)^k$ into all middle-terms for which $y_{i,j} = k$. This algorithm is particularly useful when n becomes large.

The algorithm for packing is relatively straightforward because, firstly, an accumulator is initialized with $(\!|d_{i,T-1}|\!)$ and, starting from $j = T-2$ downwards, the accumulator is exponentiated with the exponent ℓ' and the ciphertext $(\!|d_{i+j}|\!)$ is multiplied into the accumulator. This will be continued until j=0 after which $(\!|C_{\text{pack}}|\!)$ is available in the accumulator.

C. MIFE-IP based on Paillier Encryption

As discussed in Sect. II-B2, [31], [32] presented MIFE-IP based on a variation of Paillier encryption. In this paper, we consider acceleration of both the encryptions performed by the users for their respective input vectors $\mathbf{x}_i$ and the decryption performed by the server that returns the inner product $\langle \mathbf{x}, \mathbf{y} \rangle$ for a specific $\mathbf{y}$ determined by $sk_{\mathbf{y}}$. Algorithms for performing encryptions and decryptions are given in Algs. 2 and 3, respectively. In the following, we will give short introductions to the general ideas behind the algorithms and, more specifically, on the computational aspects of these algorithms. Interested readers are referred to the original publications [31], [32].

Alg. 2 first masks the user's input vector with masking values $\mathbf{u}_i$ that are part of the encryption key sk_i by performing m MAs in line 1. The purpose of this masking is to hide the result of the inner product $\langle \mathbf{x}_i, \mathbf{y}_i \rangle$ of the user i from the server when it performs the single-input FE decryption for this user's input. In the next lines, the masked input vector is encrypted with the single-input FE scheme from [31] and it includes $m + 1$ MEs, which constitute the main computational load of the algorithm, and a few MMs and MAs. The MEs are trivial to parallelize because each of them can be computed independently of each other.

Alg. 3 begins by performing single-input FE decryptions [31] for all users' ciphertexts in lines 1–7. The AHE property of Paillier encryption is used in line 5 to homo-

Algorithm 2: Encryption of MIFE-IP based on Paillier encryption [32, adapted from Figs. 1, 3 and 9]

Input: The encryption key $sk_i = (N, g, \mathbf{h}_i, \mathbf{u}_i)$ for input i consisting of the composite modulus N, a generator $g \in \mathbb{Z}^*_{N^2}$, and two vectors $\mathbf{h}_i = (h_{i,0}, \ldots, h_{i,m-1})$ with $h_{i,j} \in \mathbb{Z}_{N^2}$ and $\mathbf{u}_i = (u_{i,0}, \ldots, u_{i,m-1})$ with $u_{i,j} \in \mathbb{Z}_L$; And vector $\mathbf{x}_i = (x_{i,0}, \ldots, x_{i,m-1})$ with $x_{i,j} \in \mathbb{Z}_\ell$

Output: $\mathbf{c}_i = (c_{i,0}, \ldots, c_{i,m})$ where $c_{i,j} \in \mathbb{Z}_{N^2}$

1 $\mathbf{w} = (w_0, \ldots, w_{m-1}) \leftarrow ((x_{i,0} + u_{i,0}), \ldots, (x_{i,m-1} + u_{i,m-1})) \pmod L$
2 $r \leftarrow_R \{0, 1, \ldots, \lfloor N/4 \rfloor\}$
3 $c_{i,0} \leftarrow \mathrm{ME}(g, r, N^2)$
4 **for** $j = 0$ **to** $m - 1$ **do**
5 $\quad t_1 \leftarrow \mathrm{ME}(h_{i,j}, r, N^2)$
6 $\quad t_2 \leftarrow \mathrm{MM}(w_j, N, N^2)$
7 $\quad t_2 \leftarrow \mathrm{MA}(t_2, 1, N^2)$
8 $\quad c_{i,j+1} \leftarrow \mathrm{MM}(t_1, t_2, N^2)$
9 **return** $\mathbf{c}_i = (c_{i,0}, \ldots, c_{i,m})$

Algorithm 3: Decryption of MIFE-IP based on Paillier encryption [32, adapted from Figs. 1, 3 and 9]

Input: The decryption key $sk_{\mathbf{y}} = (N, \mathbf{y}, \mathbf{d}, z)$ for inner product $\langle \mathbf{x}, \mathbf{y} \rangle$ with the modulus N, the weight vectors $\mathbf{y} = (\mathbf{y}_0, \ldots, \mathbf{y}_{n-1})$ with $\mathbf{y}_i = (y_{i,0}, \ldots, y_{i,m-1})$ and $y_{i,j} \in \mathbb{Z}_\ell$, the vector $\mathbf{d} = (d_0, \ldots, d_{n-1})$ with $d_i \in \mathbb{Z}$, and $z \in \mathbb{Z}_L$; The ciphertexts $(\mathbf{c}_0, \ldots, \mathbf{c}_{n-1})$ where $\mathbf{c}_i = (c_{i,0}, \ldots, c_{i,m})$ and $c_{i,j} \in \mathbb{Z}_{N^2}$

Output: Inner product $r \in \mathbb{Z}_L$

1 **for** $i = 0$ **to** $n - 1$ **do**
2 $\quad r_i \leftarrow \mathrm{MI}(c_{i,0}, N^2)$
3 $\quad r_i \leftarrow \mathrm{ME}(r_i, d_i, N^2)$
4 $\quad$ **for** $j = 0$ **to** $m - 1$ **do**
5 $\quad\quad t \leftarrow \mathrm{ME}(c_{i,j+1}, y_{i,j}, N^2)$
6 $\quad\quad r_i \leftarrow \mathrm{MM}(r_i, t, N^2)$
7 $\quad r_i \leftarrow \mathrm{MS}(r_i, 1, N^2)$
8 **return** $r \leftarrow \left((N^{-1} \bmod L) \cdot \sum_{i=0}^{n-1} r_i - z \right) \bmod L$

morphically compute $x_{i,j}y_{i,j}$ via homomorphic scalar multiplications of (2) and in line 6 to compute $\sum x_{i,j}y_{i,j}$ via homomorphic additions of (1). In line 8, $N_p = N^{-1} \pmod L$ can be precomputed for each key set in the beginning. Line 8 also removes the masks by subtracting $z = \sum \langle \mathbf{u}_i, \mathbf{y}_i \rangle$, which is a part of the decryption key $sk_{\mathbf{y}}$ giving the inner product $\langle \mathbf{x}, \mathbf{y} \rangle$ as the result. The most expensive operations in Alg. 3 are the MEs in line 3; the MEs in line 5 are cheaper in practice because $y_{i,j}$ are typically small. Alg. 3 requires n modular inversions (MI) in line 2. They can be computed with just one inversion and $3(n-1)$ MMs by using the so-called Montgomery's trick [46], [47]. In that case, the inverses are computed before entering the main for loop. Because line 3 dominates in the computational cost, the most advantageous way to parallelize computation is to distribute different single-input FE decryptions (i.e., lines 3–7) to different cores.

IV. Architecture

This section describes the architecture of our flexible microcode based multi-core accelerator. It is constructed as a HW/SW codesign, where the computationally heavy long integer modular arithmetic (e.g., MMs and MEs) is performed by the HW side (FPGA) and controlling of the HW side and computation of auxiliary operations is performed by the SW side. As discussed in Sect. III, the use cases that we consider in this paper are such that they include a lot of inherent parallelism. This motivated us to design the HW side as a multi-core architecture including multiple parallel and programmable Cryptography Processor (CP) cores that are designed to have a good balance between performance and area requirements. Each CP core can be programmed with different microcodes and, hence, the architecture supports both symmetric and asymmetric processing where the CP cores process the same computations (but with different data) or different computations, respectively. The ciphertexts of Paillier encryption are large (e.g., 4096 bits with the security parameter $\kappa = 2048$) and, therefore, data communication between the CP cores and the SW side easily becomes a bottleneck for performance. Our HW/SW codesign includes a multi-level memory structure to mitigate this limitation.

A. High-Level HW/SW Codesign

Fig. 2 illustrates the high-level architecture of the HW/SW codesign which is divided into two main parts: (1) SW side and (2) HW side. The architecture is generic and can be instantiated in various programmable SoCs with minor modifications, but in the following we consider mainly an instantiation in a Xilinx Zynq-7000 all programmable SoC because we use an Avnet ZedBoard for prototyping and we will refer to the specific features of that programmable SoC whenever such a distinction is required. For Zynq-7000, the SW side (called Processing System (PS) in Xilinx terminology) consists of a dual-core ARM Cortex-A9 processor and the HW side (called Programmable Logic (PL)) is an Artix-7 FPGA.

Fig. 2 also shows the three-level Data Memory (DMEM) of the HW/SW codesign. The Level-1 DMEM (L1-DMEM) is located in each CP core and Level-2 DMEM (L2-DMEM) is shared for all CP cores, and both of them are located in the HW side. The Level-3 Memory (L3-MEM) is in the SW side and consists of both on-chip and off-chip memories (the off-chip memory is a DDR3 memory in the case of ZedBoard). The control and data communications between the SW and HW sides are organized based on the capabilities of the specific programmable SoC, and we use the Advanced Extensible Interface (AXI) ports of Zynq-7000.

1) SW Side: The SW side, which is shown in the top part of Fig. 2, is responsible for controlling all operations in the HW side and external peripherals (i.e., DDR3 memory and I/O peripherals) as well as performing computations which are not supported by the multi-core structure of the HW side. The SW side controls all actions in the HW side. This includes

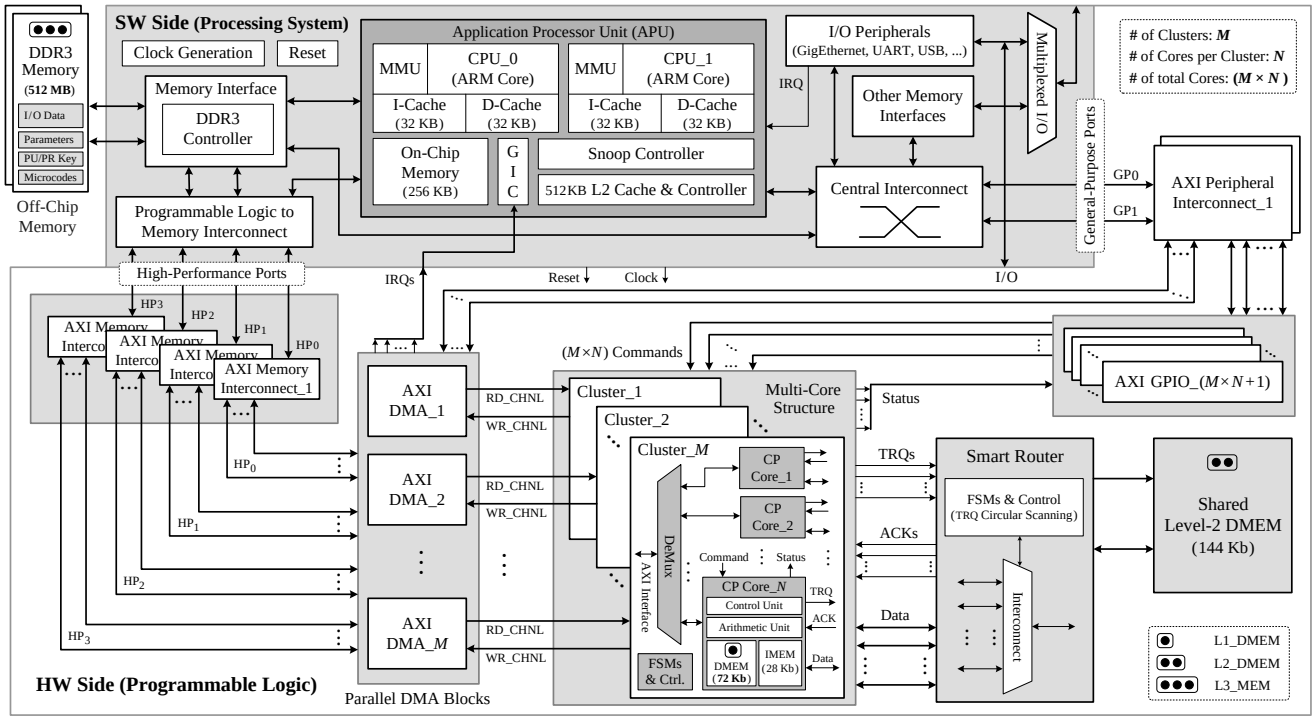

Fig. 2. High level architecture of the HW/SW codesign system. The SW and HW sides are depicted in the top and bottom parts, respectively.

sending and receiving data packets to/from the CP cores, issuing commands to the CP cores, programming all CP cores and other modules in the HW side, receiving the status of each CP core and other modules from the HW side, and making control decisions based on the received status. Thus, the SW side is responsible for high-level control of the crypto schemes.

In the case of Zynq-7000 SoCs, the two ARM processors enable parallel controlling of data packets transfer as well as command and status controlling of the CP cores and other modules in the HW side which increases throughput and efficiency. The communication between the SW and HW sides is performed via two types of ports: High Performance (HP) and General Purpose (GP) ports. There are four HP ports which are employed for high performance data transfer between the HW and SW sides as well as for programming microcodes in the CP cores. Also, there are two GP ports which are employed for commands and status transfer.

2) HW Side: The HW side, which is shown in the bottom part of Fig. 2, contains the CP cores for performing the actual computations and many supporting modules for data communication and storage as well as for commands and status transfer between the HW and SW sides. All modules in the HW side are connected in an AXI-based structure. The multi-core architecture is organized into M parallel clusters where each cluster contains N parallel CP cores, giving a total number of $M \cdot N$ CP cores in the multi-core architecture. The main challenge in this scenario is sending and receiving data packets, commands, and status between the SW side and the clusters, as well as managing the execution-flow from the SW side. The data communication between the SW side and the CP cores is done via the HP ports that connect to the AXI memory interconnect blocks which further connect to the CP cores via AXI Direct Memory Access (DMA) blocks. In Zynq-7000, we use four parallel AXI memory interconnect blocks that connect to $\lceil M/4 \rceil$ AXI DMA blocks, each connecting to one cluster of CP cores. The command and status communication is handled via the AXI peripheral interconnect blocks in the HW side. They are also used for controlling the AXI DMA blocks used for high speed data communication.

The L2-DMEM, the shared memory between the CP cores, contains a single-port RAM and a smart router for controlling the access of the CP cores to the memory. The smart router uses circular scanning (the round-robin arbiter) from the first CP core of the first cluster to the N-th CP core of the M-th cluster. The first CP core to request access that is encountered in the scan is selected and connected to the single-port RAM. This allows the CP core to transfer data between its L1-DMEM and the single-port RAM of the L2-DMEM. When the selected core with index i has finished the data transfer, the smart router will start the circular scan again from index $i + 1$.

3) Configuration of the HW/SW Codesign: The system relies on flexible high-performance data transfer between three memory levels, parallel high-performance CP cores, and parallel commands/status sending/receiving schemes. The HW/SW codesign allows high-level software developing in the SW side and efficient low-level microprogramming in the HW side, consequently, ensuring fast development of high-level control for various algorithms in the SW side and maximal performance for critical operations in the HW side.

Fig. 3 clarifies the taxonomy of different levels of memories and provides details of the command set of the CPs. The L3-MEM in the SW side is used for storing different parameters, variables, input/output data, and microcodes. The command set of the CP cores includes fields for the ID of

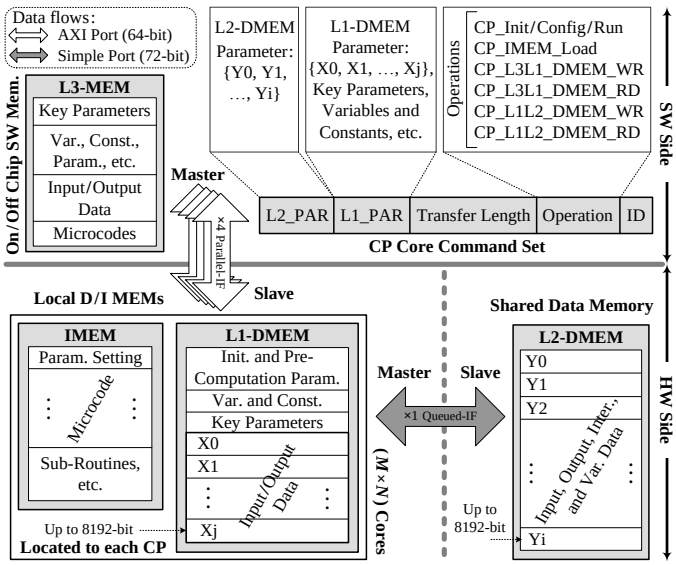

Fig. 3. Taxonomy of different memory levels and details of the CP command.

the CP core, the operation type, the transfer length between different levels of memories, the source/destination parameter in the L1-DMEM, and the source/destination parameter in the L2-DMEM. The HW side contains the L1 memories in each CP core (i.e., local L1-DMEM and Instruction Memory (IMEM)). The IMEMs store microcodes of the specific algorithm(s) and the L1-DMEMs store different parameters, intermediate variables, and input/output data. Each L1-DMEM is partitioned into partitions of up to 8192 bits denoted by X_j. Additionally, the HW side includes the shared L2-DMEM for storing input, output, and intermediate data. The L2-DMEM allows fast data transfers between the CP cores without the need to move the data between the HW and SW sides. The L2-DMEM is also partitioned to up to 8192-bit partitions called Y_i. Two main types of the data transfers are considered. Firstly, data transfers from the L3-MEM to a specific X_j in the L1-DMEMs of the CP cores, or vice versa. Secondly, data transfers from a specific X_j from a L1-DMEM to a specific Y_i in the L2-DMEM, or vice versa. Data transfers between the L2-DMEM and L3-MEM can be done via a L1-DMEM.

B. Cryptography Core

The main idea behind the design of the CP core is to have a compact, programmable, and high-performance processor for large integer modular arithmetic optimized for the resources of modern FPGAs (e.g., DSPs and BRAMs). The objective is to achieve a good trade-off between speed and area requirements for an individual CP core in order to facilitate an efficient multi-core architecture. We chose to realize the CP core based on a microprogramming architecture instead of implementing Finite State Machines (FSMs) for specific algorithms because microprogramming provides both flexibility (parameter sizes) and programmability (different algorithms) combined with a small area footprint that would be hard to achieve with FSMs.

Fig. 4(a) shows the architectural diagram of the CP core which contains an external interface unit, an arithmetic unit, a data memory unit, an address generation and control unit, and an instruction memory unit. Also, Fig. 4(b) shows the architecture details of the CP core and different external or internal interfaces and signals. The CP has a 72-bit datapath which was selected for two reasons: (1) The radix in the long integer modular arithmetic algorithms should be relatively large in order to have high performance and (2) the datapath width should match the width of the dedicated arithmetic resources and memory blocks of FPGAs (i.e., DSPs, BRAMs).

1) External Interface Unit: The external interface unit is the top module and a wrapper for other units in the CP core. The main tasks of this unit are receiving/sending command/status from/to external module(s), supporting AXI-based read and write interfaces with the SW side, performing handshakes with the smart router and supporting read and write interfaces with the shared L2-DMEM, and controlling other units of the CP.

2) Arithmetic Unit: As shown in Fig. 4(b), the arithmetic unit contains three main parts: (1) source registers, (2) arithmetic blocks, and (3) an output register. There are three main source registers (i.e., IN_REG_0/1/2) which are loaded from either the L1-DMEM (i.e., Out_1 and Out_2), the output of the arithmetic unit, zero value, and their present values. These registers provide inputs for the Modular Multiply-Add Accumulator (MMAA) and Modular Adder/Subtractor (MAS) blocks. Finally, there is an output register (i.e., OUT_REG) which takes its value from the outputs of the arithmetic blocks. The output of the arithmetic unit is an input of L1-DMEM. Also, there is an auxiliary part which extracts one bit of a vector and it is used for extracting the exponent bit of ME.

The MAS and MMAA blocks are designed to facilitate efficient computation of modular arithmetic algorithms. The MAS is a 72-bit adder/subtractor that returns the 72-bit result of addition or subtraction and a one-bit carry or borrow when given two 72-bit operands and a carry or borrow from the previous operation. The MMAA block takes three 72-bit input operands (In_0, In_1, and In_2) and computes $\text{Out}_{\text{low},j} = (\text{In}_{0,j} \cdot \text{In}_{1,j} + \text{In}_{2,j} + \text{Out}_{\text{high},(j-1)}) \bmod 2^{72}$, and $\text{Out}_{\text{high},j} = \big(\text{In}_{0,(j-1)} \cdot \text{In}_{1,(j-1)} + \text{In}_{2,(j-1)} + \text{Out}_{\text{high},(j-2)}\big) / \ 2^{72}$ where $\text{In}_{0,j}$, $\text{In}_{1,j}$, $\text{In}_{2,j}$, $\text{Out}_{\text{low},j}$, and $\text{Out}_{\text{high},j} = 0$ for $j < 0$.

Next, we clarify how the MAS and MMAA blocks are used for long integer modular arithmetic. MA and MS are computed by applying the 72-bit additions or subtractions of the MAS block iteratively. Efficient computation of long integer MMs is essential because they are the critical operations for public-key cryptography and are also used as the basis for other important modular operations (e.g., ME or inversion). Line 4 of the MM provided in Alg. 4 can be written as follows:

$$\begin{aligned} S_{i+1} &= (\sum_{j=0}^{\alpha-1} 2^{kj}\,(b_i a_j + s_{i,j}) + \sum_{j=0}^{\alpha-1} 2^{kj} q_i m_j)/2^k \\ &= (\sum_{j=0}^{\alpha} 2^{kj} r_{i,j} + \sum_{j=0}^{\alpha-1} 2^{kj} q_i m_j)/2^k \end{aligned} \tag{10}$$

where $r_{i,j} \in [0, 2^k - 1]$ and α is the number of 72-bit words required to represent the values to be multiplied (i.e., because most operations considered in this paper are computed modulo N^2 with N of size κ, then $\alpha = \lceil 2\kappa/72 \rceil$). We observe that

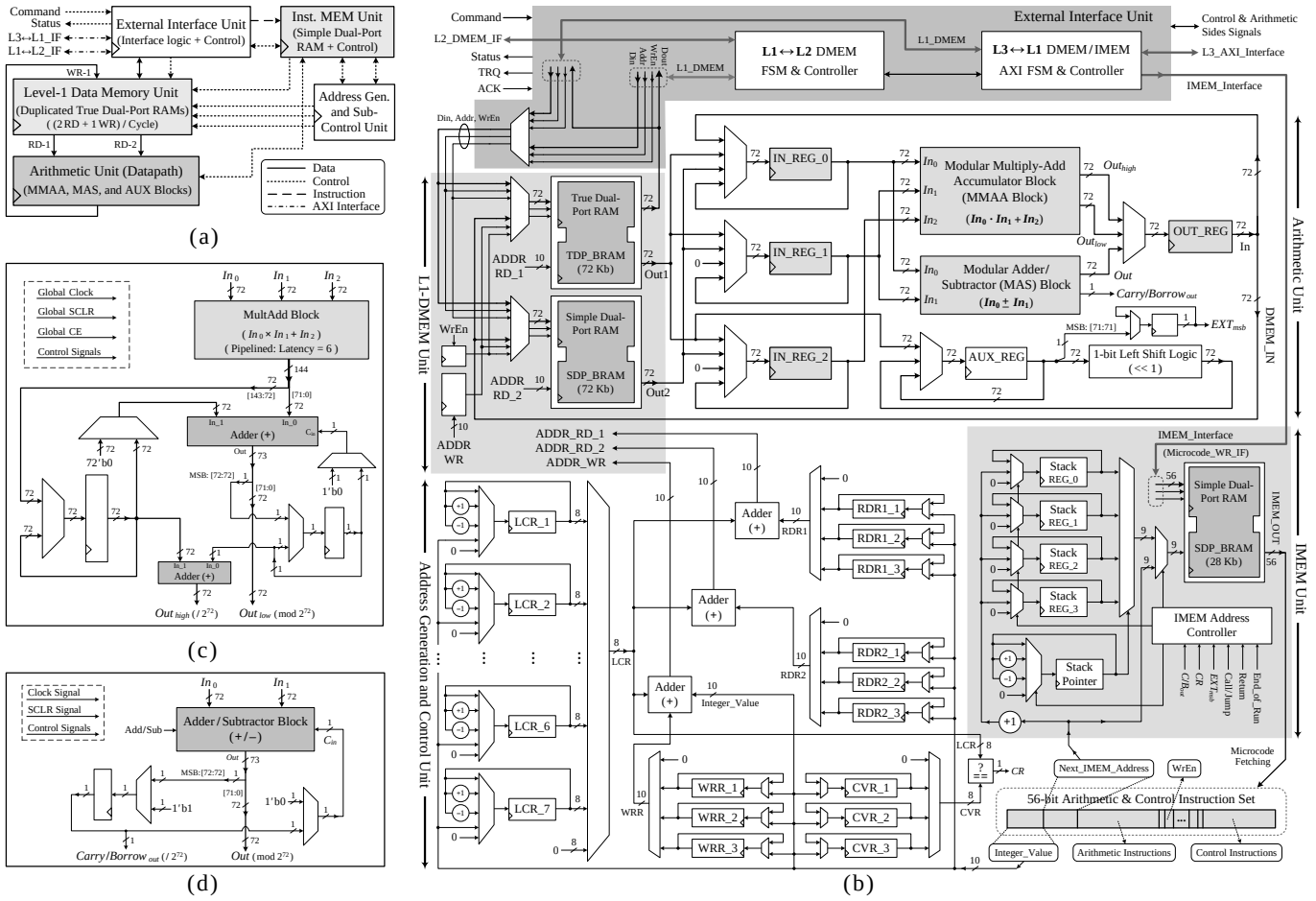

Fig. 4. (a) High-level architecture of the CP core, (b) the architectural details of the CP core, and (c–d) the structures of the MMAA and MAS blocks.

$r_{i,j} = b_i \cdot a_j + s_{i,j}$, with $j \in [0, \alpha]$ can be implemented efficiently by iterating the MMAA block in a loop over the length of the operands. The value of q_i (line 3 of Alg. 4) can be calculated based on $r_{i,0}$ as $q_i = (r_{i,0} \cdot M') \bmod 2^k$, which allows us to skip the computation of line 3 of Alg. 4. The remaining part of (10) can be expressed in the following way:

$$\begin{aligned} S_{i+1} &= (2^{k\alpha} r_{i,\alpha} + \sum_{j=0}^{\alpha-1} 2^{kj} (q_i m_j + r_{i,j}))/2^k \\ &= (2^{k\alpha}(r_{i,\alpha} + t_{i,\alpha}) + \sum_{j=0}^{\alpha-1} 2^{kj} t_{i,j})/2^k. \end{aligned} \tag{11}$$

Similarly, $t_{i,j} = q_i \cdot m_j + r_{i,j} \in [0, 2^k - 1]$ with $j \in [0, \alpha]$ can be implemented efficiently with the MMAA block.

a) MMAA Block: Fig. 4(c) shows the internal structure of the MMAA block which contains two parts: (1) a multiply-adder (i.e., MultAdd) block and (2) the complementary circuit to perform the accumulation and modular operations. The 6-level pipelined MultAdd block takes three 72-bit inputs and generates a 144-bit output. The lower part of the MultAdd output is accumulated with the previous higher part as well as the previous most significant bit of the accumulation result. The 73-bit result is divided in two part: (1) the 72-bit lower part is Out_{low} and (2) the 1-bit higher part is stored for the next accumulation. Additionally, the higher part of the MultAdd output is stored for the next accumulation. The second output of the MultAdd block (i.e., Out_{high}) is the higher part of the previous MMAA computation. Therefore, MMAA block calculates Out_{low} with a latency of six clock cycles whereas the latency for Out_{high} is seven clock cycles. The proposed accumulation method and the one clock cycle difference for Out_{low} and Out_{high} are essential for the efficiency of the long integer MM algorithm. The MMAA block supports modular

Algorithm 4: Radix-2^k Montgomery MM [44], [48]

Setting: A modulus $M > 2$ with $\gcd(M, 2) = 1$ and positive integers k and α where $4M < 2^{k\alpha}$. $R = 2^{k\alpha} \bmod M$; $M' = -M^{-1} \bmod 2^k$.
Input: $A = \sum_{j=0}^{\alpha-1} 2^{kj} a_j$, $B = \sum_{j=0}^{\alpha-1} 2^{kj} b_j$, and $M = \sum_{j=0}^{\alpha-1} 2^{kj} m_j$; $a_j, b_j, m_j \in [0, 2^k - 1]$, so that $0 \le A, B \le 2M$
Output: $S_\alpha = ABR^{-1} \bmod M$ and $0 \le S_\alpha < 2M$; $S_i = \sum_{j=0}^{\alpha-1} 2^{kj} s_{i,j}$ with $s_{i,j} \in [0, 2^k - 1]$

1 $S_0 \leftarrow 0$
2 **for** $i = 0$ **to** $\alpha - 1$ **do**
3 $\quad q_i \leftarrow (((S_i + b_i A) \bmod 2^k) M') \bmod 2^k$
4 $\quad S_{i+1} \leftarrow (S_i + q_i M + b_i A)/2^k$
5 **return** S_α

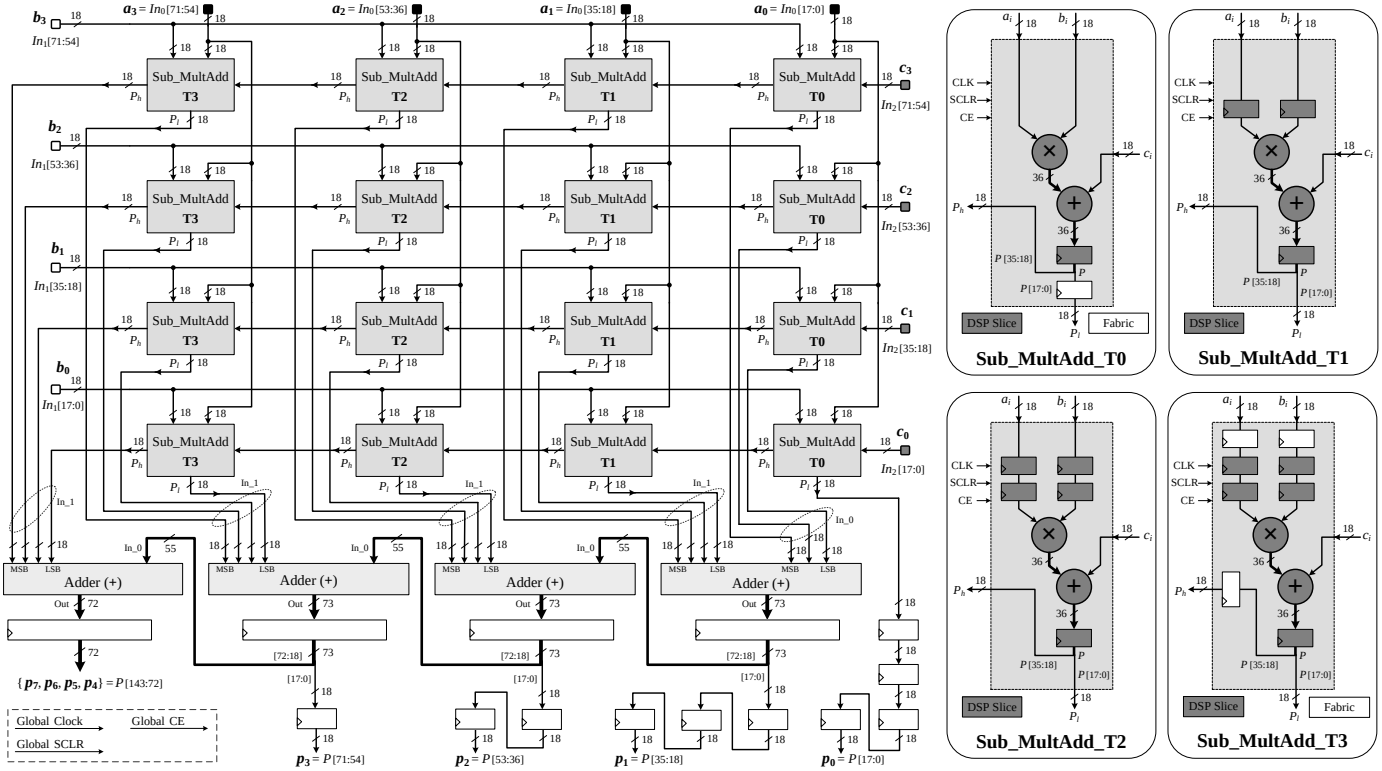

Fig. 5. Structure of the MultAdd block (6-level pipelined) and the structural details of four types of Sub_MultAdd_Tx sub-blocks.

multiply-addition with or without accumulation. The MultAdd block is the primary computation block in the datapath of each CP. It consumes most of the FPGA resources, has the highest dynamic power consumption, and also contains the critical path. In order to maximize its efficiency, it is implemented using the DSP slices. Figs. 5 show the structure of the 6-level pipelined MultAdd block and how it maps into the DSP slices.

b) MAS Block: Fig. 4(d) describes the structure of MAS block. It works in six operation modes including MA or MS with zero, one, or a carry or borrow from the previous iteration.

3) Data Memory Unit: The L1-DMEM is the local memory of each CP and is used for storing all data that is required for running an algorithm. The outputs of L1-DMEM are directly connected to the inputs of the arithmetic unit and the output of the arithmetic unit is an input to L1-DMEM unit. Based on the functionality descriptions of the MAS and MMAA arithmetic blocks given above, the most efficient way to implement long integer modular arithmetic is to read two 72-bit words and to write one 72-bit word from and to L1-DMEM simultaneously.

L1-DMEM is shown in Fig. 4(b). It contains a duplicated RAM block and some logic and registers. The first RAM is a 1024×72-bit true dual-port RAM with two independent write and read ports and the second RAM is a 1024×72-bit simple dual-port RAM with independent write and read ports. The first ports of both RAMs are used for writing an incoming data into both RAMs at the same time. The second ports of the RAMs are used for separate reads. In such an arrangement, we have L1-DMEM that can perform two reads and one write in one clock cycle. Also, the output of the first port of the first RAM is connected to the external interface unit.

4) Address Generation and Control Unit: This unit is responsible for generating read and write addresses of L1-DMEM and making control decisions for loop iterations (see Fig. 4(b)). It is constructed based on five categories of control registers: read registers 1 (RDR1_x), read registers 2 (RDR2_x), write registers (WRR_x), constant value registers (CVR_x), and loop counter registers (LCR_y), where $x \in \{1, 2, 3\}$ and $y \in \{1, \ldots, 7\}$. All registers can be initialized with an *Integer Value* from an instruction from the IMEM. Additionally, the LCR_y registers can be incremented or decremented by one and set to zero.

The read and write addresses (i.e., ADDR_RD_1/2 and ADDR_WR) for L1-DMEM are generated by adding a value from the respective register bank (i.e., from RDR1_x, RDR2_x, or WRR_x) with a value from LCR_y. The selection of specific values from these register banks is controlled with instructions from the IMEM. Different constant values (e.g., the number of iterations of a for-loop, etc.) can be saved in the CVR_x register bank and they can be compared with a value from LCR_y. The comparison result CR be used as a control signal for the IMEM (e.g., for terminating a for-loop). Details of the address generation unit can be seen in Fig. 4(b)).

5) Instruction Memory Unit: IMEM stores microcodes for algorithms that are run in the CP. The microcodes are sequences of instructions for different units of the CP. Each instruction consists of several fields such as arithmetic, control, next IMEM address, *Integer Value*, and L1-DMEM fields. These fields are responsible for applying all required commands to the corresponding units for a working cycle of the CP. IMEM has an interface via the external interface unit that

allows the SW side to load microcodes. IMEM is implemented as a 512×56-bit simple dual-port RAM. Also, it consists of a subunit for generating IMEM read addresses which accepts several control inputs from the other units and generates the next read address. IMEM supports different branch scenarios such as (un)conditional jump/call, return, start/end of program.

C. Implementation of the Use Cases in the HW/SW codesign

1) Paillier Encryption and Decryption: Paillier encryption is computed completely in the HW side by using consecutive MMs and MEs. The SW side simply controls data transfers between the SW and HW sides as well as manages the commands and status interactions.

Implementation of Paillier decryption in our HW/SW codesign consists of two phases: (1) precomputations and (2) main computations. Firstly, a set of precomputations are performed for calculating $(L(g^{\lambda} \bmod N^2))^{-1} \bmod N$ (see (4)) including ME in the HW side and an integer division and a modular inversion in the SW side. These SW side operations are performed with wolfSSL embedded library (wolfCrypt library, Ver. 3.13.0, https://www.wolfssl.com). This is a one time effort that must be done only once for every modulus N and generator g. The main computations are performed based on ME and MM in the HW side and modular division in the SW.

2) Squared Euclidean Distances: Implementation of the SEDs consists of two parts: computing the encrypted middle-terms using (6) and the final distances using (7). The first one is implemented either with the straightforward algorithm of Alg. 1(a) or the optimized algorithm of Alg. 1(b).

Alg. 1(a) is implemented as follows. Each CP is assigned for computing a distance between the user's input and the i-th entry in the server's database, where $i \in [0, n-1]$. I.e., each CP core computes consecutive MEs and MMs for a specific i and then performs the two MMs required to compute (7). Hence, all CP cores process in parallel in a row-based scheme; i.e., we consider the database as a matrix with n rows (entries) and m columns (elements of the entries). All computation of this algorithm is implemented in the HW side and the SW side performs simple control and data transfer tasks.

Alg. 1(b) is implemented as follows. The CP cores operate in a mixed column-and-row-based scheme. Firstly, they work in parallel in a column-based scheme, where columns (i.e., different iterations of the outer for-loop) are assigned to different CP cores. Each CP core computes MMs for a specific column j. Because different CP cores (different j) must contribute to the same t_i, each CP core has its own copy of t_i until it finishes the column-based processing, after which they are combined with MMs in a row-based manner. Secondly, the CPs process in parallel in a row-based scheme for computing the final results of the encrypted middle-terms using (7) similarly to the above case. Compared to Alg. 1(a), the SW side performs more control and data transfer tasks, while the HW side performs fewer operations.

3) MIFE-IP: MIFE-IP encryption of a single user's input with Alg. 2 is performed as follows. Lines 1 and 2 are performed in the SW side and the other operations are computed in the HW side. The $m+1$ elements of the ciphertext $\mathbf{c}_i$ are all computed independently and, therefore, line 3 and all m iterations of the for loop in lines 4–8 can be distributed to different CP cores, which perform them by executing MEs, MMs, and MAs. The SW side performs control and data transfer tasks, but also light computations (lines 1 and 2).

MIFE-IP decryption is performed with Alg. 3 as follows. The role of the SW side is more emphasized than in the above algorithms. Implementation of Alg. 3 consists of three phases: (1) computing the inverse (i.e., $(\prod_{i=0}^{n-1} c_{i,0})^{-1} \bmod N^2$) in the SW side based on Montgomery's trick [46], [47], (2) all the CPs in the HW side performing intensive MM, ME, and MS operations in parallel (independently for each user's input) to compute the corresponding $c_{i,0}^{-1}$ and lines 2–7 of Alg. 3, and (3) computing the inner product using line 8 in the SW side.

D. Security Model

We assume that Paillier encryption and the protocol are secure and focus on information leakage from the implementation. Furthermore, we assume that the adversary lacks physical access to the other party's computation platform (also via malware) and limit our analysis to remote timing side-channel attacks. Because PETs and FE are the primary use cases, we assume that the adversary is the other party of the protocol.

In privacy-preserving SED computation (see Fig. 1(a)), an adversarial user aims to find out the server's database $\mathbf{Y}$ and an adversarial server aims to find out either (a) the user's secret key sk or (b) the user's input $\mathbf{x}$ (note that (a) implies (b), but not vice versa). Although $\mathsf{Enc}(pk, m)$ does not use the secret key sk, it should still be CT to prevent information leakage about m via a timing channel. A CT encryption follows directly from CT MEs. It also suffices to use CT MEs to protect $\mathsf{Dec}(sk, c)$ from leaking sk or information about the plaintext. An adversarial user wants to find out $\mathbf{Y}$ by exploiting the timing of computing SEDs. The user can measure only the overall timing of computing all distances and, hence, can make estimates, e.g., on the density of $\mathbf{Y}$ and the sum of Hamming weights of all $y_{i,j}$. It is evident that this leakage is not enough to compromise $\mathbf{Y}$ or to construct $\mathbf{Y}' \approx \mathbf{Y}$ which functions similarly. Even this small leakage can be prevented by using a CT-variant of Alg. 1(a) by using CT MEs for computations with $y_{i,j}$. The cost depends particularly on the density of $\mathbf{Y}$ but also on how $y_{i,j}$ are distributed.

To accommodate protected implementations, we provide results for CT variants of Paillier encryption and decryption in Section V. MM, MA, and MS are CT by default, but ME has two versions: square-and-multiply (non-CT) and square-and-multiply-always (CT). For FE (see Fig. 1(b)), all parties perform their computations independently of any other party so there are no timing channels.

To summarize, our analysis concluded that side-channel attacks do not pose a major risk for the use cases considered in this paper but, nonetheless, we recommend to carefully analyze the risks of side-channel attacks for other use cases where our implementation may be used.

V. Results and Analysis

To get performance evaluations in real hardware, we implemented the HW/SW codesign on an Avnet ZedBoard with

TABLE I
SUMMARY OF RESOURCE REQUIREMENTS IN XILINX ZYNQ-7020

Component (#)		LUT	Reg	Slice	BRAM	DSP
Single CP core design						
External interface		73	54	28	0	0
Data-path	MMAA	690	1295	314	0	16
	MAS	152	1	58	0	0
	Other	178	387	75	0	0
DMEM		164	0	86	4	0
Control unit		327	196	111	0	0
IMEM		197	0	91	1	0
Total resource usage		1781	1933	763	5	16
		3.4%	1.8%	5.7%	3.6%	7.3%
Multi-CP core design ($M = 6$, $N = 2$)						
Mem. intercon. (3)		1976	1932	822	0	0
AXI DMA (6)		2120	3490	928	3	0
Cluster (6)	CP-1	1787	1933	761	5	16
	CP-2	1775	1933	758	5	16
	FSMs	143	109	95	0	0
Shared Mem.		377	4	200	4	0
Perph. intercon. (2)		1092	1020	568	0	0
AXI GPIOs (13)		62	237	71	0	0
System reset		17	19	13	0	0
Total resource usage		42871	53328	13096	82	192
		80.6%	50.1%	98.5%	58.6%	87.3%

TABLE II
TIMING CHARACTERISTICS FOR THE SINGLE AND MULTI-CORE DESIGN

Operation	Length (bit)	HW Latency	Tot. Time [a]	Max T-put [b]
CP Load	28 K	521	4.40 μs	—
CP Init & Config	2048	424799	3.59 ms	—
	4096	1524287	12.87 ms	—
	8192	5860174	49.48 ms	—
CP_L3L1_ DMEM_ WR/RD	2048	50	0.42 μs	6467184
	4096	82	0.70 μs	3943404
	8192	146	1.23 μs	2214789
CP_L1L2_ DMEM_ WR/RD	2048	35	0.30 μs	3384188
	4096	63	0.53 μs	1880104
	8192	120	1.01 μs	987055
MA/MS	2048	66	0.54 μs	22181808
	4096	122	1.00 μs	12000000
	8192	236	1.93 μs	6203388
MR	2048	1426	11.70 μs	1026636
	4096	4394	36.02 μs	333180
	8192	15281	125.25 μs	95796
MM	2048	2003	16.42 μs	730903
	4096	7127	54.42 μs	205416
	8192	27248	223.34 μs	53728
ME-1 [c]	2048	6159400	50.49 ms	237
	4096	43800644	359.02 ms	33
	8192	334848125	2744.66 ms	4
ME-2 [c]	2048	3086904	25.30 ms	474
	4096	21917877	179.66 ms	66
	8192	167456672	1372.60 ms	8
ME-3 [c]	2048	24081	197.39 μs	60794
	4096	85573	701.42 μs	17108
	8192	327015	2680.45 μs	4476

[a] Single CP core: (FPGA: 1×@122 MHz) and (ARM: @ 666.67 MHz).
[b] Multi-CP core: (FPGA: 12×@122 MHz) and (ARM: @ 666.67 MHz).
[c] ME-1, 2, and 3 with exponent length ℓ, $\ell/2$, and 8 bits and Hamming weights $\ell/2$, $\ell/4$, and 4 bits, respectively, where $\ell \in \{2048, 4096, 8192\}$.

a low-cost Xilinx Zynq-7020 xc7z020clg484-1 including a dual-core ARM Cortex A9 and an Artix-7 FPGA. For the SW side, we used C++ and Xilinx Software Development Kit (SDK) for developing software for a Real-Time Operating System (RTOS). For the HW side, we used Verilog (HDL) and Vivado 16.3. The resource requirements are summarized in Table I. The maximum clock frequencies for the FPGA and ARM are 122 and 667 MHz, respectively. Based on Vivado, the total power consumption of the chip is about 3.2W. All reported results are final post-place&route results and validated with real hardware, unless mentioned otherwise. Also, in Table II, we report the detailed timings of different operations including initialization of CP cores, data transfers between memory levels, and modular arithmetic.

A. Performance of the Use Cases

As mentioned, our HW/SW codesign is primarily implemented for acceleration of use cases that are enabled by AHE and Paillier encryption in particular. We evaluated the performance of such use cases on our HW/SW codesign by focusing on privacy-preserving computation of SEDs and MIFE-IP. Timing and Peak Memory Usage (PMU) results are collected in Table III. PMUs are reported for the HW (i.e., L1-DMEM and L2-DMEM) and SW (i.e., L3-MEM) sides separately. Both Paillier encryption and decryption require roughly 180 ms with $\kappa = 2048$. This timing is the time that it takes to encrypt or decrypt a single plaintext or ciphertext in a single CP. The decryption timing is for the main computation only, as the precomputation is performed offline (see Sect. IV-C). The multi-core architecture can process different operations in parallel and, thus, the HW/SW codesign can process about 66 encryptions or decryptions per second.

We study the use case of SEDs with different database sizes and densities. We consider three sizes including small (S) $n = 32$ and $m = 16$, medium (M) $n = 128$ and $m = 32$, and large (L) $n = 512$ and $m = 64$, where n is the number of entries in the server's database and m is the length of the user's input and each database entry. In practice, databases are often sparse in the sense that most of the values are zeroes (e.g., [25] reports that 85.4 % of the database values are zeroes), which allows to skip them in computations. We study the effects of sparseness by considering two database densities (DBD) 25 % and 100 %. For both cases, the database values are represented with one byte (i.e., $y_{i,j} \in \mathbb{Z}_\ell$ with $\ell = 256$). As expected, Alg. 1(b) quickly becomes faster than Alg. 1(a), which is superior only for small and sparse databases. However, Alg. 1(b) has a larger memory footprint, especially, for 'large' datasets (e.g., the database L with density 100 % has the PMUs of 592 and 16730 KB in the SW side for Alg. 1(a) and 1(b), respectively).

Comparisons with software implementations are difficult because exact counterparts are not available or publications provide only high-level timings including also other operations. Nevertheless, the delays are typically in the order of seconds with parameter sizes comparable to Table III [17], [20]. Evans et al. [17] reported distance computation timings of 1.68 s and 6.82 s with $m = 640$ for $n = 128$ and $n = 512$, respectively, for a fingerprint matching application in an Intel Xeon E5504 at 2.0 GHz. Nieminen [26] presented an implementation of privacy-preserving indoor localization where SEDs (comparable to Alg. 1(b)) for a database with $n = 150$, $m = 50$, $y_{i,j} \in \mathbb{Z}_{16}$, and density 20 % required 334.55 ms in a quad-core Intel Xeon E5-2697 at 2.7 GHz. Because the soft-

TABLE III
PERFORMANCE CHARACTERISTICS OF THE PAILLIER AND USE CASES

Operation ($\kappa = 2048$)		Latency (# of clocks) FPGA	Latency (# of clocks) ARM	Time (ms)	PMU [a] (KB) HW	PMU [a] (KB) SW
Paillier cryptosystem (single core: w/ one CP core)						
Enc		22010717	23205	180.45	12	14
Enc (CT)		29313492	23650	240.31	12	14
Dec		21920763	694161	180.72	10	17
Dec (CT)		29195087	696265	240.35	10	17
SEDs #1 (Straightforward Alg. 1 (a) + (7)) (multi-core: w/ 12 CP cores)						
DBD = 25 %	S [b]	1273472	270504	10.84	139	67
	M [b]	8858942	1190748	74.40	139	175
	L [b]	70889244	5855616	589.84	139	592
DBD = 100 %	S	4529635	460122	37.82	139	67
	M	33146342	1885572	27.520	139	175
	L	258970564	11191440	2139.50	139	592
SEDs #2 (Optimized Alg. 1 (b) + (7)) (multi-core: w/ 12 CP cores)						
DBD = 25 %	S	3813697	949416	32.68	121	118
	M	6426916	2536638	56.48	121	663
	L	17333581	12236328	160.43	121	4770
DBD = 100 %	S	4074144	1401582	35.50	121	307
	M	8392168	5607654	77.20	121	2158
	L	32821507	37348116	325.05	121	16730
Ciphertext Packing of (8) (multi-core: w/ 12 CP cores)						
$\ell' = 2^{32}$	S	7293968	92820	59.93	12	28
	M	29870654	339456	245.35	12	76
	L	29884840	396474	245.55	46	268
MIFE-IP (multi-core: w/ 12 CP cores)						
Enc Alg. 2	S' [b]	43863516	270504	359.94	84	48
	M' [b]	65795274	399126	539.90	126	82
	L' [b]	131590548	784992	1079.79	126	146
Dec Alg. 3	S'	23448922	32799936	241.40	44	117
	M'	50071208	33667140	460.92	132	823
	L'	170167033	40629303	1455.76	132	6343

[a] The HW side includes $1 \times$ L1 + L2 and $12 \times$ L1 + L2 for single CP core and multi-CP cores designs, respectively (and 168 KB in total); and the SW side (L3) includes 256 KB on-chip memory and 512 MB off-chip memory (DDR3).
[b] Small (S / S'): $n = 32 / 4$, $m = 16$, Medium (M / M'): $n = 128 / 16$, $m = 32$, and Large (L / L'): $n = 512 / 64$, $m = 64$.

ware timings in the literature are scattered and lack details, we used our own Python 2.7.16 program on Intel Core i5-6267U at 2.9 GHz for comparisons. It computes the operations of [26] in 157.47 ms and is actually faster than [26]. The 'large' database with DBD 100% takes 3913.12 ms. Consequently, we get an approximately 12 times speedup with Zynq-7020.

We study also the use case of MIFE-IP with datasets of three different sizes including small (S'): $n = 4$ and $m = 16$, medium (M'): $n = 16$ and $m = 32$, and large (L'): $n = 64$ and $m = 64$, where n is the number of users and m is the length of each user's input. The results show that encryption timing grows linearly with m (or more precisely $\lceil m/12 \rceil$) whereas decryption timing has a linear dependency on n (or more precisely $\lceil n/12 \rceil$). Decryption needs more memory than encryption. The main reason is that encryption is performed separately for each user, while decryptions computed by the server need to handle ciphertexts from all users simultaneously. Comparisons with software implementations are even more difficult in the case of MIFE-IP because, to the best of our knowledge, implementation results for MIFE-IP based on Paillier encryption are not available[1]. This is understandable

[1]Software implementations of FE are available in Github (github.com/fentec-project/) but this scheme is not currently supported.

TABLE IV
COMPARISON OF FPGA-BASED 1024-BIT ME IMPLEMENTATIONS

Reference / Device	F_{max} MHz	Area (slice)	BRAM / DSP [a]	Time (ms)	Flex	A·D [b]
[45] / Virtex-4	150	6633	— / —	11.95	No	79.3
[49] / Virtex-5	401	12716	— / —	0.92	No	11.7
[50] / Virtex-7	485	1060	— / 26	2.33	No	2.5
[51] / Virtex-5	385	4060	— / —	2.03	No	8.2
[52] / Virtex-5	345	3218	— / —	3.18	No	10.2
[53] / Virtex-4	400	3937	7 / 17	1.71	No	6.7
[54] / Zynq	69	249	5 / 22	15.76	No	3.9
This work [c]	122	763	5 / 16	7.79	Yes [d]	5.9

[a] The specific details of the blocks vary between different FPGAs.
[b] A·D stands for Area-Delay-Product (slices $\times$ seconds).
[c] Single CP design on Zynq-7020: FPGA @122MHz, ARM @ 667MHz.
[d] Flexible (for parameter sizes), programmable (various algorithms).

because FE is still an emerging concept and mostly theoretical papers without implementation results are available. Hence, we used our own Python 2.7.16 program also in this case. E.g., encryption and decryption with the 'large' database took 6.88 s and 8.61 s, respectively; i.e., 6.3 and 5.9 times speedups.

B. Comparisons

To the best of our knowledge, the only work currently available in the literature about FPGA acceleration of Paillier encryption was published by San et al. [7] in 2016. Their accelerator focused on speeding up the computation time for Paillier encryption and decryption, but the use of the accelerator for actual applications was only shortly studied by providing results for a privacy-preserving set intersection protocol with a small parameter set (smaller than our 'small' cases in Table III). Their implementation is not flexible in any sense as separate implementations are needed for encryption and decryption and for all different security parameters κ. When $\kappa = 2048$, their implementations for encryption and decryption require 4497 slices and 45 DSPs and 3868 slices and 27 DSPs, respectively, in a high-end Xilinx Virtex-7 FPGA. They compute encryption and decryption in 105.15 ms (34,176,060 clock cycles at 325 MHz) and 110.24 ms (34,396,228 clock cycles at 312 MHz), respectively. Our HW/SW codesign system computes in 180.451 ms and 180.720 ms, respectively, on a low-cost FPGA with only 763 slices and 16 DSPs even with just one CP. Hence, we have a much better trade-off between area and speed. Also, our work is the first one in the literature to exploit the inherent parallelism of use cases of AHE and that can be used for accelerating various use cases based on Paillier encryption with different kind of parameters.

Despite the lack of related work on acceleration of AHE, there are multiple works on FPGA acceleration of ME. At least in certain settings, they could also be used for Paillier computations. Because ME is a central operation also for our HW/SW codesign, we provide a comparison in Table IV. We chose security parameter $\kappa = 1024$ to maximize the coverage of related work. Our CP core has comparable performance and area requirements despite being the only truly flexible design in the comparison (supports up to 8192-bit modular arithmetic) and being implemented on a low-cost FPGA.

C. Implementation on a High-End Programmable SoC

Our HW/SW codesign is flexible also in the sense that it can be implemented on different platforms besides the Zynq-7020 considered above. To demonstrate this and to estimate its efficiency on a modern high-performance programmable SoC, we also compiled it for a Xilinx Zynq UltraScale+ MPSoC ZU9EG chip (i.e., xczu9eg-2ffvb1156e) featuring a quad-core ARM Cortex-A53 processor running up to 1.5GHz in the SW side and a 16nm FinFET+ based FPGA in the HW side. Such a programmable SoC allows a significantly more powerful instance of our HW/SW codesign to be implemented in a single chip. In particular, we were able to fit 96 CP cores (8 clusters with 12 CPs in each cluster) into the FPGA by using 33391 slices (97.46 %), 504 BRAMs (55.26 %), and 1536 DSPs (60.95 %). Also, the maximum frequency increased to 193 MHz. The following estimates are based on post-place&route results, but are not measured from real hardware.

The increased clock frequencies give a direct speedup (approximately by a factor 1.6), but a much larger advantage comes from the increased number of CP cores. We estimate that the 'large' cases for SEDs in Table III would be computed in 16.391 ms (2,670,078 FPGA clocks and 3,867,056 ARM clocks) and 30.547 ms (4,718,188 FPGA clocks and 9,207,342 ARM clocks) for database densities of 25 % and 100 %, respectively. These represent speedups of factors 9.8 and 10.6, respectively, compared to Zynq-7020. For MIFE-IP, we estimate that encryption and decryption for the 'large' cases in Table III would require 113.932 ms (21,948,158 FPGA clocks and 580,928 ARM clocks) and 169.627 ms (28,420,506 FPGA clocks and 33,899,055 ARM clocks), respectively. These estimates give speedup factors 9.5 and 8.6 for encryption and decryption, respectively. HW/SW communication may result in a small decrease in practical speedups, but we expect it to be minor because the delays are dominated by computation. Furthermore, we emphasize that such speedups were obtained despite the fact that even the 'large' MIFE-IP cases in Table III are too small to fully utilize the parallel processing capabilities of the 96 CP cores and use only 64 of them. To summarize, implementing our HW/SW codesign on a high-end programmable SoC gives a major advantage and enables very fast computation of even large datasets.

VI. Conclusions

We presented an efficient HW/SW codesign on a programmable SoC for accelerating applications of Paillier encryption. The implementation uses a microcode-based multi-core design optimized for efficient computation of long integer modular arithmetic. The microcode-based reprogrammability allows to use the same architecture for various different applications and the multiple cores allow exploiting the inherent parallelism of the use cases. We instantiated our HW/SW codesign in an Avnet ZedBoard with Xilinx Zynq-7020 programmable SoC and demonstrated its operation on real hardware. We also investigated the efficiency of the HW/SW codesign in a high-end Xilinx UltraScale+ programmable SoC. We evaluated two types of use cases of Paillier encryption: privacy-preserving computation of SEDs and MIFE-IP. Both of them represent the first hardware based acceleration results for such applications. In particular, the implementation of the MIFE-IP is, to the best of our knowledge, the first published implementation of that specific FE scheme on any platform and among the very first implementations of FE altogether (including software). Our work showed that HW/SW codesigns are capable of delivering good performance for such use cases, consequently, increasing their practical attractiveness.

There are also possible further directions to explore in improving the performance of our HW/SW codesign. E.g., performance could be increased with an expanse in memory consumption by performing different types of precomputations (e.g., using a window-based ME algorithm). Other possibility is to replace Paillier encryption with another AHE based on large integer modular arithmetic such as Damgård-Geisler-Krøigaard (DGK) [55], [56] or Decisional Diffie–Hellman (DDH) (e.g., [31]). Such optimizations and experiments can be done via microcode updates without changing the current architecture. More profound optimizations include changing the architecture from a homogeneous architecture (all CP cores are similar) to a heterogeneous architecture where different cores are specialized for specific tasks (e.g., core for modular inversions to accelerate the MIFE-IP decryption).

Finally, we see that a lot more research is required in accelerating FE schemes. Our work was the very first one in this domain and there are many possible directions to continue. We focused on the rather limited functionality of computing linear functions (inner products) but many practical use cases require more expressive functions such as quadratic functions. There are efficient schemes for quadratic functions available (e.g., [57]), but they require a larger set of cryptographic primitives including, especially, pairings. Pairings are required also for function hiding FE [32], [40], [42], where the function to be evaluated is hidden from the server, or decentralized FE [41], which removes the need for a trusted key authority. Consequently, future work includes extending our work to support cryptographic pairings and these additional features. There are also FE schemes based on different types of cryptographic assumptions including lattice-based schemes which may be more efficient in practice, but would require completely different kind of a computation architecture.

Acknowledgments

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 780108 (FENTEC).

References

[1] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," *Found. Secur. Comput.*, vol. 4, no. 11, pp. 169–180, 1978.

[2] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *STOC*, 2009, pp. 169–178.

[3] A. Sahai and B. Waters, "Fuzzy identity-based encryption," in *EUROCRYPT*, ser. LNCS, vol. 3494. Springer, 2005, pp. 457–473.

[4] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano, "Public key encryption with keyword search," in *EUROCRYPT*, ser. LNCS, vol. 3027. Springer, 2004, pp. 506–522.

[5] D. Boneh, A. Sahai, and B. Waters, "Functional encryption: Definitions and challenges," in *TCC*, ser. LNCS, vol. 6597. Springer, 2011, pp. 253–273.

[6] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *EUROCRYPT*, ser. LNCS, vol. 1592. Springer, 1999, pp. 223–238.

[7] I. San, N. At, I. Yakut, and H. Polat, "Efficient Paillier cryptoprocessor for privacy-preserving data mining," *Secur. Commun. Netw.*, vol. 9, no. 11, pp. 1535–1546, 2016.

[8] W. Wang and X. Huang, "FPGA implementation of a large-number multiplier for fully homomorphic encryption," in *ISCAS*, 2013, pp. 2589–2592.

[9] Y. Doröz, E. Öztürk, and B. Sunar, "Evaluating the hardware performance of a million-bit multiplier," in *DSD*, 2013, pp. 955–962.

[10] C. Moore, N. Hanley, J. McAllister, M. O'Neill, E. O'Sullivan, and X. Cao, "Targeting FPGA DSP slices for a large integer multiplier for integer based FHE," in *WAHC*, ser. LNCS, vol. 7862. Springer, 2013, pp. 226–237.

[11] T. Pöppelmann, M. Naehrig, A. Putnam, and A. Macias, "Accelerating homomorphic evaluation on reconfigurable hardware," in *CHES*, ser. LNCS, vol. 9293. Springer, 2015, pp. 143–163.

[12] S. Sinha Roy, K. Järvinen, F. Vercauteren, V. Dimitrov, and I. Verbauwhede, "Modular hardware architecture for somewhat homomorphic function evaluation," in *CHES*, ser. LNCS, vol. 9293. Springer, 2015, pp. 164–184.

[13] Y. Doröz, E. Öztürk, E. Savas, and B. Sunar, "Accelerating LTV based homomorphic encryption in reconfigurable hardware," in *CHES*, ser. LNCS, vol. 9293. Springer, 2015, pp. 185–204.

[14] E. Öztürk, Y. Doröz, E. Savaş, and B. Sunar, "A custom accelerator for homomorphic encryption applications," *IEEE Trans. Comput.*, vol. 66, no. 1, pp. 3–16, Jan. 2017.

[15] S. Sinha Roy, K. Järvinen, J. Vliegen, F. Vercauteren, and I. Verbauwhede, "HEPCloud: An FPGA-based multicore processor for FV somewhat homomorphic function evaluation," *IEEE Trans. Comput.*, vol. 67, no. 11, pp. 1637–1650, Nov. 2018.

[16] ISO, "IT security techniques – encryption algorithms – part 6: Homomorphic encryption," International Organization for Standardization, Geneva, Switzerland, ISO/IEC 18033-6:2019, 2019.

[17] D. Evans, Y. Huang, J. Katz, and L. Malka, "Efficient privacy-preserving biometric identification," in *NDSS*, 2011.

[18] M. Barni, T. Bianchi, D. Catalano, M. Di Raimondo, R. Donida Labati, P. Failla, D. Fiore, R. Lazzeretti, V. Piuri, F. Scotti, and A. Piva, "Privacy-preserving fingercode authentication," in *MM&Sec*. ACM, 2010, pp. 231–240.

[19] Z. Erkin, M. Franz, J. Guajardo, S. Katzenbeisser, I. Lagendijk, and T. Toft, "Privacy-preserving face recognition," in *PETS*, ser. LNCS, vol. 5672. Springer, 2009, pp. 235–253.

[20] A.-R. Sadeghi, T. Schneider, and I. Wehrenberg, "Efficient privacy-preserving face recognition," in *ICISC*, ser. LNCS, vol. 5984. Springer, 2010, pp. 229–244.

[21] F. D. Garcia and B. Jacobs, "Privacy-friendly energy-metering via homomorphic encryption," in *STM*, ser. LNCS, vol. 6710. Springer, 2011, pp. 226–238.

[22] R. Lu, X. Liang, X. Li, X. Lin, and X. S. Shen, "EPPA: An efficient and privacy-preserving aggregation scheme for secure smart grid communications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 9, pp. 1621–1631, Sep. 2012.

[23] M. Jawurek and F. Kerschbaum, "Fault-tolerant privacy-preserving statistics," in *PETS*, ser. LNCS, vol. 7384. Springer, 2012, pp. 221–238.

[24] H. Li, L. Sun, H. Zhu, X. Lu, and X. Cheng, "Achieving privacy preservation in WiFi fingerprint-based localization," in *INFOCOM*. IEEE, 2014, pp. 2337–2345.

[25] Z. Yang and K. Järvinen, "The death and rebirth of privacy-preserving WiFi fingerprint localization with Paillier encryption," in *INFOCOM*. IEEE, 2018, pp. 1223–1231.

[26] R. Nieminen, "Privacy-preserving indoor localization with Paillier encryption and garbled circuits," Master's thesis, Aalto University, 2018.

[27] X. Yi, A. Bouguettaya, D. Georgakopoulos, A. Song, and J. Willemson, "Privacy protection for wireless medical sensor data," *IEEE Trans. Depend. Sec. Comput.*, vol. 13, no. 3, pp. 369–380, 2016.

[28] R. Zhang, J. Zhang, Y. Zhang, J. Sun, and G. Yan, "Privacy-preserving profile matching for proximity-based mobile social networking," *IEEE J. Sel. Areas Commun.*, vol. 31, no. 9, pp. 656–668, 2013.

[29] P. Gasti and K. B. Rasmussen, "Privacy-preserving user matching," in *WPES*. ACM, 2015, pp. 111–120.

[30] A. O'Neill, "Definitional issues in functional encryption," Cryptology ePrint Archive, Report 2010/556, 2010.

[31] S. Agrawal, B. Libert, and D. Stehlé, "Fully secure functional encryption for inner products, from standard assumptions," in *CRYPTO*, ser. LNCS, vol. 9816. Springer, 2016, pp. 333–362.

[32] M. Abdalla, D. Catalano, D. Fiore, R. Gay, and B. Ursu, "Multi-input functional encryption for inner products: Function-hiding realizations and constructions without pairings," in *CRYPTO*, ser. LNCS, vol. 10991. Springer, 2018, pp. 597–627.

[33] Q. Wang, M. He, M. Du, S. S. M. Chow, R. W. F. Lai, and Q. Zou, "Searchable encryption over feature-rich data," *IEEE Trans. Depend. Sec. Comput.*, vol. 15, no. 3, pp. 496–510, May 2018.

[34] A. C.-C. Yao, "How to generate and exchange secrets," in *SFCS*. IEEE, 1986, pp. 162–167.

[35] S. D. Gordon, J. Katz, F.-H. Liu, E. Shi, and H.-S. Zhou, "Multi-input functional encryption," Cryptology ePrint Archive, Report 2013/774, 2013.

[36] S. Goldwasser, S. D. Gordon, V. Goyal, A. Jain, J. Katz, F.-H. Liu, A. Sahai, E. Shi, and H.-S. Zhou, "Multi-input functional encryption," in *EUROCRYPT*, ser. LNCS, vol. 8441. Springer, 2014, pp. 578–602.

[37] B. Waters, "A punctured programming approach to adaptively secure functional encryption," in *CRYPTO*, ser. LNCS, vol. 9216. Springer, 2015, pp. 678–697.

[38] S. Garg, C. Gentry, S. Halevi, M. Raykova, A. Sahai, and B. Waters, "Candidate indistinguishability obfuscation and functional encryption for all circuits," *SIAM J. Comput.*, vol. 45, no. 3, pp. 882–929, 2016.

[39] M. Abdalla, R. Gay, M. Raykova, and H. Wee, "Multi-input inner-product functional encryption from pairings," in *EUROCRYPT*, ser. LNCS, vol. 10210. Springer, 2017, pp. 601–626.

[40] A. Bishop, A. Jain, and L. Kowalczyk, "Function-hiding inner product encryption," in *ASIACRYPT*, ser. LNCS, vol. 9452. Springer, 2015, pp. 470–491.

[41] J. Chotard, E. Dufour Sans, R. Gay, D. H. Phan, and D. Pointcheval, "Decentralized multi-client functional encryption for inner product," in *ASIACRYPT*, ser. LNCS, vol. 11273. Springer, 2018, pp. 703–732.

[42] P. Datta, R. Dutta, and S. Mukhopadhyay, "Functional encryption for inner product with full function privacy," in *PKC*, ser. LNCS, vol. 9614. Springer, 2016, pp. 164–195.

[43] P. L. Montgomery, "Modular multiplication without trial division," *Math. Comput.*, vol. 44, no. 170, pp. 519–521, 1985.

[44] T. Blum and C. Paar, "Montgomery modular exponentiation on reconfigurable hardware," in *ARITH*. IEEE, 1999, pp. 70–77.

[45] ——, "High-radix Montgomery modular exponentiation on reconfigurable hardware," *IEEE Trans. Comput.*, vol. 50, no. 7, pp. 759–764, 2001.

[46] P. L. Montgomery, "Speeding the Pollard and elliptic curve methods of factorization," *Math. Comput.*, vol. 48, no. 177, pp. 243–264, 1987.

[47] H. Shacham and D. Boneh, "Improving SSL handshake performance via batching," in *CT-RSA*, ser. LNCS, vol. 2020. Springer, 2001, pp. 28–43.

[48] H. Orup, "Simplifying quotient determination in high-radix modular multiplication," in *ARITH*. IEEE, 1995, pp. 193–199.

[49] A. Rezai and P. Keshavarzi, "High-throughput modular multiplication and exponentiation algorithms using multibit-scan–multibit-shift technique," *IEEE Trans. VLSI Syst.*, vol. 23, no. 9, pp. 1710–1719, 2014.

[50] I. San and N. At, "Improving the computational efficiency of modular operations for embedded systems," *J. Syst. Architect.*, vol. 60, no. 5, pp. 440–451, 2014.

[51] G. D. Sutter, J.-P. Deschamps, and J. L. Imaña, "Modular multiplication and exponentiation architectures for fast RSA cryptosystem based on digit serial computation," *IEEE Trans. Ind. Electron.*, vol. 58, no. 7, pp. 3101–3109, 2010.

[52] T. Wu, S. Li, and L. Liu, "Fast, compact and symmetric modular exponentiation architecture by common-multiplicand Montgomery modular multiplications," *Integration*, vol. 46, no. 4, pp. 323–332, 2013.

[53] D. Suzuki, "How to maximize the potential of FPGA resources for modular exponentiation," in *CHES*, ser. LNCS, vol. 4727. Springer, 2007, pp. 272–288.

[54] L. Rodríguez-Flores, M. Morales-Sandoval, R. Cumplido, C. Feregrino-Uribe, and I. Algredo-Badillo, "Compact FPGA hardware architecture for public key encryption in embedded devices," *PloS one*, vol. 13, no. 1, pp. 1–21, 2018.

[55] I. Damgård, M. Geisler, and M. Krøigaard, "Efficient and secure comparison for on-line auctions," in *ACISP*, ser. LNCS, vol. 4586. Springer, 2007, pp. 416–430.

[56] ——, "A correction to "efficient and secure comparison for on-line auctions"," Cryptology ePrint Archive, Report 2008/321, 2008.

[57] C. E. Z. Baltico, D. Catalano, D. Fiore, and R. Gay, "Practical functional encryption for quadratic functions with applications to predicate encryption," in *CRYPTO*, ser. LNCS, vol. 10401. Springer, 2017, pp. 67–98.